Design Rationale Sprint 2 - Alex Ung
Explain two key classes (not interfaces) that you have included in your design and provide your reasons why you decided to create these classes. Why was it not appropriate for them to be methods?
Classes:

MovementManager
The MovementManager class is very important in my design for the fiery dragons game. Anything that requires movement will be passed through this class. This class has 3 methods in total excluding the getInstance method. So in the fiery Dragon board game, movement can be made by Dragon Tokens. The first thing that must be considered before a DragonToken can move, is can the DragonToken legally move to a new position according to the game rules.
The game rules specify that a Dragon Token can move to a new position as long as:

- There is not another Dragon Token on the new position
- If the movement is backwards, is the DragonToken inside of their cave already?

The second method is named updatePosition, so this handles all of the nitty gritty things when it comes to updating the player's position. It does not update the UI, but it handles the internal game logic. It determines how many positions to move depending on if the DragonToken is still in their cave or not and it takes into account backwards movements.

For when DragonCards are implemented, the movementManager will leverage a third method within itself which is the isMatch() method. This will be where the logic for handling matching dragon cards and square creatures which will determine whether the movement can be made.

The reason why I did not have this class as a set of methods inside the player or the board for example, is because I think that there are too many responsibilities related to the movement logic for the game. If I had this class merged with another class like the DragonToken class, the DragonToken class would have too many responsibilities and hence breaking the single responsibility principle. By having this movement manager class I have been able to encapsulate everything related to movement in this class. This will also make it easier if I want to add different kinds of movements in the future because there is a standalone class that manages it which would respect the open/closed principle.

BoardArray class
The board array class is the glue for the creation of the board. The BoardArray holds everything related to the creation of the board. I wanted to have the logic for the board all in one place which makes it easier to access different parts of the board. So for the BoardArray class, it will add VolcanoCards and shuffle them to be placed on the board, it will also add the dragon cards to the board. The reason why this class cannot be a method is because it would add another responsibility to another class and I wouldn't want to have access to information that is not relevant to the actual board itself when I am accessing the board from another

class. This class aims to be in line with the single responsibility and is a part of the chain of responsibilities design pattern that will be discussed later on in the rationale.

Explain two key relationships in your class diagram, for example, why is something an aggregation not a composition?

> volcanoCard -> Square Composition relationship
> So in the case of my volcanoCard class, my interpretation of the game makes me think that because volcano cards are pieces that are placed by the BoardArray, why should the BoardArray have to take care of the squares that are on each volcanoCard. So I've made the VolcanoCard class have a composition relationship with the square class. In the context of the game itself, if there are no volcano cards, then there wouldn't be any squares. So it wouldn't be aggregation because a square cannot exist independently of a volcano card.
>
> DragonToken -><- DragonTokenPanel
> So this relationship between the two classes represent a circular association. In this scenario I do not think that this is a bad design purely because in order to move dragon tokens i'll need to have access to the specific panel that the dragonToken has been allocated. The same goes for the dragonTokenPanel, in the case of the UI, there is a

Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?
I used it, but only in applications where I saw extensions to be applicable. For example, I used inheritance on the DragonToken class because a DragonToken is an actor and there could definitely be extensions that could involve the introduction of actors that could do different things. To give an example, it would be possible to have multiple different types of dragonTokens that have different abilities like a Mischievous wizard token that can activate obstacles or give riddles to a player to set them back a few squares.

The other case where I used inheritance was with the action class. For now there is really only one action that a player can take which is flipping a dragon card but what if there was an introduction of landing on a square which allows the player to choose to pick up a card (from a deck of cards) that acts like a mystery card. This reduces a lot of the repeated code that is seen with classes that have similar attributes and methods.

Explain how you arrived at two sets of cardinalities, for example, why 0..1 and why not 1…2?
MovementManager Class

If you have used any Design Patterns in your design, explain why and where you have applied them. If you have not used any Design Patterns, list at least 3 known Design Patterns and justify why they were not appropriate to be used in your design.
I have used singleton
I have used chain of responsibility

Facade